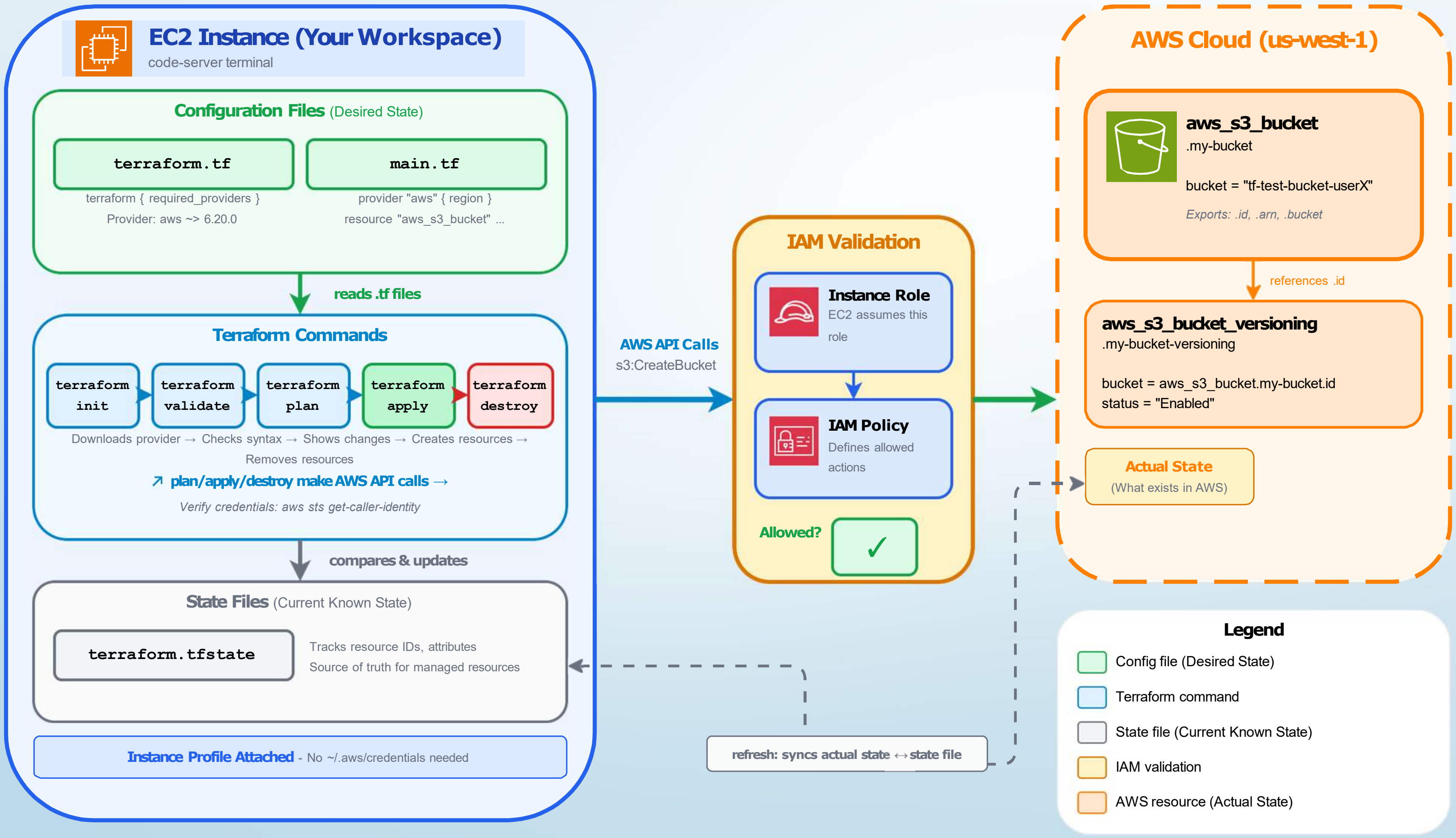
Module 3: Core Workflow and Lifecycle

Terraform level 1 - Day 1 - **Module 3**

Module 3: Terraform Core Workflow


EC2 Instance (Your Workspace)
code-server terminal
Configuration Files (Desired State)
terraform.tf
terraform { required_providers }
Provider: aws ~> 6.20.0
main.tf
provider "aws" { region }
resource "aws_s3_bucket" ...
reads .tf files
Terraform Commands
terraform init
terraform validate
terraform plan
terraform apply
terraform destroy
Downloads provider → Checks syntax → Shows changes → Creates resources → Removes resources
↗ plan/apply/destroy make AWS API calls →
Verify credentials: aws sts get-caller-identity
compares & updates
State Files (Current Known State)
terraform.tfstate
Tracks resource IDs, attributes
Source of truth for managed resources
Instance Profile Attached - No ~/.aws/credentials needed
AWS API Calls
s3:CreateBucket
IAM Validation
Instance Role
EC2 assumes this role
IAM Policy
Defines allowed actions
Allowed?
✓
refresh: syncs actual state ↔ state file
AWS Cloud (us-west-1)
aws_s3_bucket
.my-bucket
bucket = "tf-test-bucket-userX"
Exports: .id, .arn, .bucket
references .id
aws_s3_bucket_versioning
.my-bucket-versioning
bucket = aws_s3_bucket.my-bucket.id
status = "Enabled"
Actual State
(What exists in AWS)
Legend
Config file (Desired State)
Terraform command
State file (Current Known State)
IAM validation
AWS resource (Actual State)

0. Check if terraform has been installed:

Type in the terminal and press enter:

```
terraform -version
```

1. [AWS Access] Have access to resources via specific provider (AWS)

Let's check if AWS credentials has been configured:

```
aws sts get-caller-identity
```

A few sources of AWS credentials:

- local static credentials assigned to a specific AWS user, credentials are located ~/.aws/credentials
- AWS Role associated with an EC2 instance (an instance profile)
- AWS credentials are establishing specific identity: AWS User, AWS Role
- Permissions are defined by attaching an IAM Policy to specific identity

In our case, you're working on an EC2 instance that is using specific AWS Role,
and what you AWS resources you can access from this instance are defined by IAM Policy attached to this role.

2. [Terraform AWS Config] Configure this provider (AWS) in terraform

Without telling terraform to use specific provider (AWS) terraform don't know what to do:

```
cd
mkdir module3-project
cd module3-project
terraform init
```

Creating configuration for AWS provider: terraform.tf

```
terraform {

  required_providers {
    aws = {
      source    = "hashicorp/aws"
      version   = "~> 6.20.0"
    }
  }

  required_version = "~> 1.13.5"
}
```

Note about specifying versions:

In both cases we use "~>" symbol which means upgrade to newest patch version (the last digit in version format major.minor.patch), but no further.

For AWS provider it's "~> 6.20.0"
and when new version will be
available it will upgrade to 6.20.1
and 6.20.2
and so on, but not to 6.21.

For Terraform itself it's "~> 1.13.5" and the symbol works in the same way: upgrade to 1.13.6, 7, 8 and so on, but not to 1.14

The reason behind using "~>" symbol is to prevent unexpected breaking changes in your configuration with new minor or major versions. You can consider those changes at some point in a separate environment.

3. [Terraform init] Create/Initialize the default workspace

Terraform will download specific provider and create a local file **.terraform.lock.hcl** to store information about provider's version.

This file should be version controlled, so that everyone on the team will use the same provider's version.

Let's try to create workspace once again:

```
terraform init
```

One new directory was created to store provider's binary and one file to store provider's version that we're currently using:

```
ls -ld .terraform*
```

4. [Terraform validate] Check for configuration errors.

```
terraform validate
```

5. [Defining AWS resource] Let's create some simple resource:

First we need to specify the region where our resources will be created: main.tf

```
provider "aws" {
  region                      = "us-west-1"

  # IMPORTANT: Change 'userX' to your assigned student ID (user1, user2, etc.)
  default_tags {
    tags = {
      owner = "userX"
    }
  }
}
```

Then let's create an s3 bucket, replace your initials, with the actual initials since the name of the bucket has to be unique:

```
resource "aws s3 bucket" "my-bucket" {
```

6. [Terraform plan] See what would change when we apply current configuration

Currently you haven't defined any infrastructure, so there will be no changes:

```
terraform plan
```

7. [Optional][Terraform fmt] Format configuration files according to canonical style

This command will remove extra spaces from both .tf files:

```
terraform fmt
```

8. [Terraform apply] Once again look at the plan, if looks good, confirm to realize it:

```
terraform apply
```

You can verify that a new bucket has been created:

```
terraform state list
```

or with specific attribute details:

```
terraform show
```

```
export BUCKET="tf-test-bucket-userX"
aws s3 ls | grep $BUCKET
```

Internally, terraform created two new files in current directory, those files are storing information about current state of resources in this workspace (we will talk about state in

If you try to run plan again, terraform will check if there are

any changes to resource configuration (compare what's on AWS to what's inside the state file), you've just created the bucket, so no changes are needed:

```
terraform plan
```

9. Change the bucket configuration

Add at the end of main.tf:

```
resource "aws_s3_bucket_versioning" "my-bucket-versioning" {
    bucket = aws_s3_bucket.my-bucket.id
    versioning_configuration {
        status = "Enabled"
    }
}
```

Now, let's check the potential changes:

```
terraform plan
```

Let's enable versioning on the bucket:

```
terraform apply
```

10.1 [Drift I] Dealing with unexpected changes in configuration outside terraform, so called drift:

Let's change bucket's configuration using aws cli (not using terraform):

```
aws s3api put-bucket-versioning \
  --bucket $BUCKET \
  --versioning-configuration Status=Suspended
```

Is it now disabled (suspended)?:

```
aws s3api get-bucket-versioning --bucket $BUCKET
```

Now, let's see if terraform detect this change:

```
terraform plan
```

Terraform wants now to revert this change back to the configuration defined in terraform files, let's do it:

```
terraform apply
```

Bucket configuration is now back to the original configuration:

```
aws s3api get-bucket-versioning --bucket $BUCKET
```

10.2 [Drift II] Sometimes you might want to update your terraform configuration to reflect unexpected changes:

Let's again change bucket's configuration using aws cli (not using terraform):

```
aws s3api put-bucket-versioning \
  --bucket $BUCKET \
  --versioning-configuration Status=Suspended
```

Let's check if terraform detects this change:

```
terraform plan
```

Terraform wants to revert the change back to "Enabled". But what if we want to keep the change that was made? Simply update the configuration to match it.

Change the configuration in main.tf to match the current state:

```
resource "aws_s3_bucket_versioning" "my-bucket-versioning" {
    bucket = aws_s3_bucket.my-bucket.id
    versioning_configuration {
        status = "Suspended"
    }
}
```

Now, when you run plan again, terraform will see that the configuration matches the actual AWS resource state, so no changes are needed:

- terraform plan

- You'll see "No changes. Your infrastructure matches the configuration." Terraform automatically refreshed the state during the plan, so there's nothing to apply.

- **11. [Warning][Terraform destroy] Terraform will show you which resource will be deleted forever:**

- Confirm if it's an s3 bucket that you've just created:

- terraform destroy